AVF Control Number: AVF-VSR-023
                     SZT-AVF-023

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 88051111.09136
Straessle Systementwicklungs GmbH u. Co.KG
CADMUS/VADS
Version 5.5
PCS CADMUS 9032

Completion of On-Site Testing:
88-05-11

Prepared By:
IABG m.b.H., Dept SZT
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

DTIC
ELECTE
FEB 1 4 1989
H

Ada is a registered trademark of the United States Government
(Ada Joint Program Office).

89 2 13 086

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Compiler Validation Summary Report: Straessle Systementwicklungs GmbH u. Co.KG, CADMUS/VADS Version 5.5, PCS CADMUS 9032 (Host and Target). (880511 I1. 09136)		5. TYPE OF REPORT & PERIOD COVERED 11 May 1988 to 11 May 1989
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) IABG, Ottobrunn, Federal Republic of Germany.		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS IABG, Ottobrunn, Federal Republic of Germany.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		12. REPORT DATE 11 May 1988
		13. NUMBER OF PAGES 44 p.
14. MONITORING AGENCY NAME & ADDRESS *(If different from Controlling Office)* IABG, Ottobrunn, Federal Republic of Germany.		15. SECURITY CLASS *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
CADMUS/VADS Version 5.5, Straessle Systementwicklungs GmbH u. Co.KG, IABG, PCS CADMUS 9032 under Munix, V5.2/32 (Host and Target), ACVC 1.9.

DD FORM 1473   EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73   S/N 0102-LF-014-6601

Ada   Compiler Validation Summary Report:

Compiler Name: CADMUS/VADS
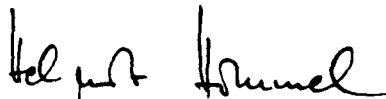Compiler Version: Version 5.5

Certificate Number: 88051111.09136

Host:
    PCS CADMUS 9032 under
    Munix,
    V5.2/32

Target:
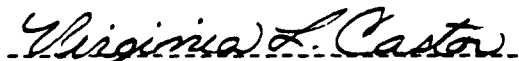   Same as host

Testing Completed 88-05-11 Using ACVC 1.9

This report has been reviewed and is approved.

IABG m.b.H., Dept.SZT
Dr. H. Hummel
Einsteinstrasse 20
8012 Ottobrunn
Federal Republic of Germany

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA   22311

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC   20301

2

CONTENTS

Accession For

NTIS GRA&I	☑
DTiC TAB	☐
Unannounced	☐
Justification	

By

Distribution/

Availability Codes

| Dist | Avail and/or Special |

A-1

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an
Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the
  compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by
  the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed
  by the Ada Standard

Testing of this compiler was conducted by IABG mbH according to procedures
established by the Ada Joint Program Office and administered by the Ada
Validation Organization (AVO). On-site testing was completed 88-05-11 at
Straessle Systementwicklung in D-6143 Lorsch.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may
make full and free public disclosure of this report. In the United States,
this is provided in accordance with the "Freedom of Information Act" (5
U.S.C. #552). The results of this validation apply only to the computers,
operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not
represent or warrant that all statements set forth in this report are
accurate and complete, or that the subject compiler has no nonconformities
to the Ada Standard other than those presented. Copies of this report are
available to the public from:

>   Ada Information Clearinghouse
>   Ada Joint Program Office
>   OUSDRE
>   The Pentagon, Rm 3D-139 (Fern Street)
>   Washington DC 20301-3081

or from:

>   IABG m.b.H., Dept SZT
>   Einsteinstrasse 20
>   8012 Ottobrunn
>   Federal Republic of Germany

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA   22311

1.3   REFERENCES

1.   Reference Manual for the Ada Programming Language,
     ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2.   Ada Compiler Validation Procedures and Guidelines, Ada Joint
     Program Office, 1 January 1987.

3.   Ada Compiler Validation Capability Implementers' Guide, SofTech,
     Inc., December 1986.

4.   Ada Compiler Validation Capability User's Guide, December 1986.

INTRODUCTION

1.4   DEFINITION OF TERMS

ACVC   The Ada Compiler Validation Capability. The set of Ada
               programs that tests the conformity of an Ada compiler to the
               Ada programming language.

Ada   An Ada Commentary contains all information relevant to the
Commentary   point addressed by a comment on the Ada Standard.   These
               comments are given a unique identification number having the
               form AI-ddddd.

Ada Standard   ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant   The agency requesting validation.

AVF   The Ada Validation Facility.   The AVF is responsible for
               conducting compiler validations according to procedures
               contained in the Ada Compiler Validation Procedures and
               Guidelines.

AVO   The Ada Validation Organization.   The AVO has oversight
               authority   over   all   AVF   practices   for   the purpose of
               maintaining a uniform process for validation of Ada
               compilers.   The AVO provides administrative and technical
               support for Ada validations to ensure consistent practices.

Compiler   A processor for the Ada language. In the context of this
               report, a compiler is any language processor, including
               cross-compilers, translators, and interpreters.

Failed test   An ACVC test for which the compiler generates a result that
               demonstrates nonconformity to the Ada Standard.

Host   The computer on which the compiler resides.

Inapplicable   An ACVC test that uses features of the language that a
test   compiler is not required to support or may legitimately
               support in a way other than the one expected by the test.

Passed test   An ACVC test for which a compiler generates the expected
               result.

Target   The computer for which a compiler generates code.

Test   A program that checks a compiler's conformity regarding a
               particular feature or a combination of features to the Ada
               Standard. In the context of this report, the term is used to
               designate a single test, which may comprise one or more
               files.

Withdrawn   An ACVC test found to be incorrect and not used to check
test   conformity to the Ada Standard.   A test may be incorrect

because it has an invalid test objective, fails to meet its
test objective, or contains illegal or erroneous use of the
language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC
contains both legal and illegal Ada programs structured into six test
classes: A, B, C, D, E, and L. The first letter of a test name identifies
the class to which it belongs. Class A, C, D, and E tests are executable,
and special program units are used to report their results during
execution. Class B tests are expected to produce compilation errors.
Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled
and executed. There are no explicit program components in a Class A test
to check semantics. For example, a Class A test checks that reserved words
of another language (other than those already reserved in the Ada language)
are not treated as reserved words by an Ada compiler. A Class A test is
passed if no errors are detected at compile time and the program executes
to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class
B tests are not executable. Each test in this class is compiled and the
resulting compilation listing is examined to verify that every syntax or
semantic error in the test is detected. A Class B test is passed if every
illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and
executed. Each Class C test is self-checking and produces a PASSED,
FAILED, or NOT APPLICABLE message indicating the result when it is
executed.

Class D tests check the compilation and execution capacities of a compiler.
Since there are no capacity requirements placed on a compiler by the Ada
Standard for some parameters--for example, the number of identifiers
permitted in a compilation or the number of units in a library--a compiler
may refuse to compile a Class D test and still be a conforming compiler.
Therefore, if a Class D test fails to compile because the capacity of the
compiler is exceeded, the test is classified as inapplicable. If a Class D
test compiles successfully, it is self-checking and produces a PASSED or
FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED,
or FAILED message when it is compiled and executed. However, the Ada
Standard permits an implementation to reject programs containing some
features addressed by Class E tests during compilation. Therefore, a Class
E test is passed by a compiler if it is compiled successfully and executes
to produce a PASSED message, or if it is rejected by the compiler for an
allowable reason.

INTRODUCTION

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: CADMUS/VADS, Version 5.5

ACVC Version: 1.9

Certificate Number: 88051111.09136

Host Computer:

	Machine:	PCS CADMUS 9032
	Operating System:	Munix V5.2/32
	Memory Size:	8 MByte

Target Computer:

Same as host

CONFIGURATION INFORMATION

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of
a compiler in those areas of the Ada Standard that permit implementations
to differ. Class D and E tests specifically check for such implementation
differences.   However, tests in other classes also characterize an
implementation. The tests demonstrate the following characteristics:

. Capacities.

   The compiler correctly processes tests containing loop statements
   nested to 65 levels, block statements nested to 65 levels, and
   recursive procedures separately compiled as subunits nested to 17
   levels.   It correctly processes a compilation containing 723
   variables in the same declarative part. (See tests D55A03A..H (8
   tests), D56001B, D64005E..G (3 tests), and D29002K.)

. Universal integer calculations.

   An implementation is allowed to reject universal integer
   calculations having values that exceed SYSTEM.MAX_INT.   This
   implementation processes 64 bit integer calculations. (See tests
   D4A002A, D4A002B, D4A004A, and D4A004B.)

. Predefined types.

   This implementation supports the additional predefined types
   Short_Float,   Short_Integer   and Tiny_Integer in the package
   STANDARD (See tests B86001CP, B86001CR and B86001D.)

. Based literals.

   An implementation is allowed to reject a based literal with a
   value exceeding SYSTEM.MAX_INT during compilation, or it may raise
   NUMERIC_ERROR or CONSTRAINT_ERROR during execution.   This
   implementation raises NUMERIC_ERROR during execution. (See test
   E24101A.)

. Expression evaluation.

   Apparently no default initialization expressions for record
   components are evaluated before any value is checked to belong to
   a component's subtype. (See test C32117A.)

   Assignments for subtypes are performed with the same precision as
   the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

. Rounding.

The method used for rounding to integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round to even. (See test C4A014A.)

CONFIGURATION INFORMATION

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or
CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds
STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this
implementation:

Declaration of an array type or subtype declaration with more than
SYSTEM.MAX_INT components raises no exception. (See test
C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type
with INTEGER'LAST + 2 components. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type
with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST
raises NUMERIC_ERROR when the array type is declared. (See test
C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST
components raises NUMERIC_ERROR when the length of a dimension is
calculated and exceeds INTEGER'LAST. (See test C52104Y.)

A null array with one dimension of length greater than
INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either
when declared or assigned. Alternatively, an implementation may
accept the declaration. However, lengths must match in array
slice assignments. This implementation raises NUMERIC_ERROR when
the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears
to be evaluated in its entirety before CONSTRAINT_ERROR is raised
when checking whether the expression's subtype is compatible with
the target's subtype. In assigning two-dimensional array types,
the expression does not appear to be evaluated in its entirety
before CONSTRAINT_ERROR is raised when checking whether the
expression's subtype is compatible with the target's subtype.
(See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept
or reject an incomplete type with discriminants that is used in an
access type definition with a compatible discriminant constraint.
This implementation accepts such subtype indications. (See test
E38104A.)

In assigning record types with discriminants, the expression
appears to be evaluated in its entirety before CONSTRAINT_ERROR is
raised when checking whether the expression's subtype is

compatible with the target's subtype.   (See test C52013A.)

. Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype.   (See test E43211B.)

. Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported.   (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported.   (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported.   (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported.   (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported.   (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported.   (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported.   (See tests A39005E and C87B62C.)

Record representation clauses are supported.   (See test A39005G.)

CONFIGURATION INFORMATION

Length clauses with SIZE specifications for derived integer types
are supported. (See test C87B62A.)

. Pragmas.

The pragma INLINE is supported for procedures. The pragma INLINE
is supported for functions. (See tests LA3004A, LA3004B, EA3004C,
EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained
array types and record types with discriminants without defaults.
(See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO can be instantiated with unconstrained array
types and record types with discriminants without defaults. (See
tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See
tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for
DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO.
(See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for
SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last
element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be
created in OUT_FILE mode, and can be created in IN_FILE mode.
(See test EE3102C.)

More than one internal file can be associated with each external
file for text I/O for both reading and writing. (See tests
CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external
file for sequential I/O for both reading and writing. (See tests
CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external
file for direct I/O for both reading and writing. (See tests
CE2107F..I (5 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file can be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file can be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

. Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was
tested, 27 tests had been withdrawn because of test errors. The AVF
determined that 225 tests were inapplicable to this implementation. All
inapplicable tests were processed during validation testing except for 201
executable tests that use floating-point precision exceeding that supported
by the implementation. Modifications to the code, processing, or grading
for 23 tests were required to successfully demonstrate the test objective.
(See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable
conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	110	1049	1630	17	18	46	2870
Inapplicable	0	2	223	0	0	0	225
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

TEST INFORMATION

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	190	500	540	245	166	98	142	326	137	36	234	3	253	2870
Inapplicable	14	72	134	3	0	0	1	1	0	0	0	0	0	225
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of
this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features
that a compiler is not required by the Ada Standard to support. Others may
depend on the result of another test that is either inapplicable or
withdrawn.   The applicability of a test to an implementation is considered
each time a validation is attempted. A test that is inapplicable for one
validation attempt is not necessarily inapplicable for a subsequent
attempt. For this validation attempt, 225 tests were inapplicable for the
reasons indicated:

- C35702B uses LONG_FLOAT, which is not supported by this compiler.

- The following tests use LONG_INTEGER, which is not supported by
  this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	

C45632C   B52004D   C55B07A   B55B09C

. C45531M, C45531N, C45532M, and C45532N use fine 48 bit fixed point
  base types which are not supported by this compiler.

. C45531O, C45531P, C45532O, and C45532P use coarse 48 bit fixed
  point base types which are not supported by this compiler.

. C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by
  this new definition in this implementation and the test cannot be
  executed since the package REPORT is dependent on the package
  TEXT_IO.

. The following 201 tests require a floating-point accuracy that
  exceeds the maximum of 15 digits supported by this implementation:

        C24113L..Y (14 tests)   C35705L..Y (14 tests)
        C35706L..Y (14 tests)   C35707L..Y (14 tests)
        C35708L..Y (14 tests)   C35802L..Z (15 tests)
        C45241L..Y (14 tests)   C45321L..Y (14 tests)
        C45421L..Y (14 tests)   C45521L..Z (15 tests)
        C45524L..Z (15 tests)   C45621L..Z (15 tests)
        C45641L..Y (14 tests)   C46012L..Z (15 tests)

. C96005B requires the range of type DURATION to be different from
  those of its base type; in this implementation they are the same.

3.6   TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code,
processing, or evaluation in order to compensate for legitimate
implementation behavior. Modifications are made by the AVF in cases where
legitimate implementation behavior prevents the successful completion of an
(otherwise) applicable test. Examples of such modifications include:
adding a length clause to alter the default size of a collection; splitting
a Class B test into subtests so that all errors are detected; and
confirming that messages produced by an executable test demonstrate
conforming behavior that wasn't anticipated by the test (such as raising
one exception instead of another).

Modifications were required for 1 Class A tests and 22 Class B tests.

Test A39005G was modified by inserting PRAGMA PACK (STATE_MASK) in the
declarative part to enable the compiler to process successfully a later
record representation clause. This modification was done in accordance
with an AVO message dated 88-05-18. The modified test produced the message
==PASSED== and was graded passed.

TEST INFORMATION

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A	B24204A	B24204B	B24204C	B33301A
B37201A	B38003A	B38003B	B38009A	B38009B
B44001A	B64001A	B67001A	B67001B	B67001C
B67001D	B91003B	B95001A	B95007B	B97102A
BC1303F	BC3005B			

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the CADMUS/VADS was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the CADMUS/VADS using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a PCS CADMUS 9032 operating under Munix, V5.2/32 .

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were not included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled on the PCS CADMUS 9032, and all executable tests were linked and run. Results were printed from the host computer.

The compiler was tested using command scripts provided by Straessle Systementwicklungs GmbH u. Co.KG and reviewed by the validation team. The compiler was tested using all default settings.

Tests were compiled, linked, and executed (as appropriate) using a single host computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Straessle Systementwicklungs GmbH u.   Co.KG in
D-6143 Lorsch, Germany and was completed on 88-05-11.

APPENDIX A

DECLARATION OF CONFORMANCE

Straessle Systementwicklungs GmbH u. Co.KG has
submitted the following Declaration of Conformance
concerning the CADMUS/VADS.

DECLARATION OF CONFORMANCE

Compiler Implementor: Straessle Systementwicklungs GmbH u. Co.KG
Ada Validation Facility: IABG m.b.H., Dept SZT
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: CADMUS/VADS
Base Compiler Version: Version 5.5
Host Architecture ISA: PCS CADMUS 9032
Target Architecture ISA: Same as host

Implementor's Declaration

I, the undersigned, representing Straessle Systementwicklungs GmbH u.
Co.KG, have implemented no deliberate extensions to the Ada Language
Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration.
I declare that Straessle Systementwicklungs GmbH u. Co.KG is the owner of
record of the Ada language compiler(s) listed above and, as such, is
responsible for maintaining said compiler(s) in conformance to
ANSI/MIL-STD-1815A. All certificates and registrations for Ada language
compiler(s) listed in this declaration shall be made only in the owner's
corporate name.

ppa. G. Kitzinger   Date: 6.7.88
Straessle Systementwicklungs GmbH u. Co.KG

Owner's Declaration

I, the undersigned, representing Straessle Systementwicklungs GmbH u.
Co.KG, take full responsibility for implementation and maintenance of the
Ada compiler(s) listed above, and agree to the public disclosure of the
final Validation Summary Report. I further agree to continue to comply
with the Ada trademark policy, as defined by the Ada Joint Program Office.
I declare that all of the Ada language compilers listed, and their
host/target performance, are in compliance with the Ada Language Standard
ANSI/MIL-STD-1815A.

ppa. G. Kitzinger   Date: 6.7.88
Straessle Systementwicklungs GmbH u. Co.KG

A-2

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to
implementation-dependent pragmas, to certain machine-depen-
dent conventions as mentioned in chapter 13 of the Ada
Standard, and to certain allowed restrictions on represen-
tation clauses. The implementation-dependent characteris-
tics of the CADMUS/VADS, Version 5.5, are described in the
following pages B-2 ... B-14 which are taken from the Im-
plementation Reference Manual and discuss topics in Appen-
dix F of the Ada Standard. Implementation-specific portions
of the package STANDARD are also included in this appendix.

18.4 Pragmas and Their Effects

Each of this implementation's pragmas is briefly described here; additional information on some of them is found under discussions of particular language constructs.

pragma BUILT_IN may be used in some parts of the code for TEXT_IO, MACHINE_CODE, UNCHECKED_CONVERSION, UNCHECKED_DEALLOCATION, and lower level support packages in STANDARD. It is reserved for use by Verdix and is not directly accessible to the user.

pragma CONTROLLED is recognized by the implementation, but has no effect in the current release.

pragma ELABORATE is implemented as described in Appendix B of the RM.

pragma EXTERNAL_NAME allows the user to specify a *link_name* for an Ada variable or subprogram so that the object can be referenced from other languages.

See also
[Implementation Reference], Interface Programming

pragma IMPLICIT_CODE specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF) and is used only within the declarative part of a machine code procedure.

See also
[Implementation Reference], Machine Code Insertions and *Using VADS, Machine Code Insertions* for details.

pragma INLINE is implemented as described in Appendix B of the RM with the addition that recursive calls can be expanded with the pragma up to the maximum depth of 8. Warnings are produced for too-deep nestings or for bodies that are not available for inline expansion.

pragma INLINE_ONLY when used in the same way as pragma INLINE, indicates to the compiler that the subprogram must *always* be inlined (very important for some code procedures.). This pragma also suppresses the generation of a callable version of the routine which saves code space.

pragma INTERFACE supports calls to C and FORTRAN language functions with an optional linker name for the subprogram. The Ada specifications can be either functions or procedures. All parameters must have mode IN.

See also
[Implementation Reference], Interface Programming

pragma INTERFACE_OBJECT allows variables defined in another language to be referenced directly in Ada, replacing all occurrences of *variable_name* with an external reference to *link_name* in the object file.

See also
[Implementation Reference], Interface Programming

pragma LIST is implemented as described in Appendix B of the RM.

pragma MEMORY_SIZE is recognized by the implementation, but has no effect in the current release.

pragma NO_IMAGE suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image.

pragma OPTIMIZE is recognized by the implementation, but has no effect in the current release.

For code optimization options, see also *VADS Reference,* ada, –O.

pragma PACK will cause the compiler to minimize gaps between components in the representation of composite types. For arrays, components will only be packed to bit sizes corresponding to powers of 2 (if the field is smaller than STORAGE_UNIT bits). Objects larger than a single STORAGE_UNIT are packed to the nearest STORAGE_UNIT.

pragma PAGE is implemented as described in Appendix B of the RM. It is also recognized by the source code formatting tool a.pr.

pragma PRIORITY is implemented as described in Appendix B of the RM.

pragma SHARE_CODE provides for the sharing of object code between multiple instantiations of the same generic procedure or package body. A 'parent' instantiation is created and subsequent instantiations of the same types can share the parent's object code code, reducing program size and compilation times. The name pragma SHARE_BODY may be used instead of SHARE_CODE with the same effect.

pragma SHARED is recognized by the implementation, but has no effect in the current release.

pragma STORAGE_UNIT is recognized by the implementation, but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling package SYSTEM with altered values.

pragma SUPPRESS is supported in the single parameter form. The pragma applies from the point of occurrence to the end of the innermost enclosing block. DIVISION_CHECK cannot be suppressed. The double parameter form of the pragma with a name of an object, type, or subtype is recognized, but has no effect in the current release.

pragma SYSTEM_NAME is recognized by the implementation, but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the file system.a from the STANDARD library can be copied to a local VADS library and recompiled there with new values.

18.5 Implementation–defined Attribute: 'REF

VADS provides one implementation–defined attribute, 'REF. There are two forms of use for this attribute, X'REF and SYSTEM.ADDRESS'REF(N). X'REF is used only in machine code procedures while SYSTEM.ADDRESS'REF(N) can be used anywhere to convert an integer expression to an address.

18.5.1 X'REF

The attribute generates a reference to the entity to which it is applied.

In X'REF, X must be either a constant, variable, procedure, function, or label. The attribute returns a value of the type MACHINE_CODE.OPERAND and may only be used to designate an operand within a code–statement.

The instruction generated by the code–statement in which the attribute occurs may be preceded by additional instructions needed to facilitate the reference (i.e., loading a base register). If the declarative section of the procedure contains pragma IMPLICIT_CODE (OFF), a warning will be generated if additional code is required.

References may also cause run time checks to be generated. pragma SUPPRESS may be used to eliminate these checks.

```
CODE_1'(JSR, PROC'REF);
CODE_2'(MOVE_L, X.ALL(Z)'REF, D0);
```

See also
Using VADS, Machine Code Insertions.

18.5.2 SYSTEM.ADDRESS'REF(N)

The effect of this attribute is similar to the effect of an unchecked conversion from integer to address. However, SYSTEM.ADDRESS'REF(N) should be used instead in the following listed circumstances, and in these circumstances, N must be static.

In SYSTEM.ADDRESS'REF(N), SYSTEM.ADDRESS must be the type SYSTEM.ADDRESS. N must be an expression of type UNIVERSAL_INTEGER. The attribute returns a value of type SYSTEM.ADDRESS, which represents the address designated by N.

- within any of the run time configuration packages
  Use of unchecked conversion within an address clause would require the generation of elaboration code, but the configuration packages are not elaborated.

- in any instance where N is greater than INTEGER'LAST
  Such values are required in address clauses which reference the upper portion of memory. To use unchecked conversion in these instances would require that the expression be given as a negative integer.

- to place an object at an address, use the 'REF attribute
  The *integer_value*, in the example below, is converted to an address for use in the address clause representation specification. The form avoids UNCHECKED_CONVERSION and is also useful for 32-bit unsigned addresses.

```
--place an object at an address
for object use at ADDRESS'REF (integer_value)

--to use unsigned addresses
for VECTOR use at SYSTEM.ADDRESS'REF(16#808000d0#);
TOP_OF_MEMORY : SYSTEM.ADDRESS := SYSTEM.ADDRESS'REF(16#FFFFFFFF#);
```

18.9 Representation Specifications

Representation Clauses — VADS supports bit-level representation clauses.

pragma PACK — Objects and components are packed to the nearest power of two bits. VADS does not define any additional representation pragmas.

Length Clauses — VADS supports all representation clauses.

Enumeration Representation Clauses — Enumeration representation clauses are supported.

Record Representation Clauses — Representation specifications are based on the target machine's word, byte, and bit order numbering so that VADS is consistent with machine architecture manuals for both 'big-endian' and 'little-endian' machines. Bits within a STORAGE_UNIT are also numbered according to the target machine manuals. It is not necessary for a user to

understand the default layout for records and other aggregates since fine control over the layout is obtained by the use of record representation specifications. It is then possible to align record fields correctly with structures and other aggregates from other languages by specifying the location of each element explicitly. Note that bit fields are numbered opposite the ordering for bits within a byte on M68000 family processors. Bit fields use the numbering specified for the MC68020 bit extraction instructions. The 'FIRST_BIT and 'LAST_BIT attributes can be used to construct bit manipulation code that is applicable to differently bit-numbered systems.

Refer to the M68000 Family addressing and bit numbering illustration on the next page.
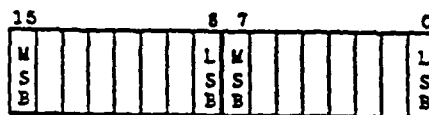
M68000 Family Addressing and Bit Numbering*

M
S
B   most significant bit

L
S
B   least significant bit

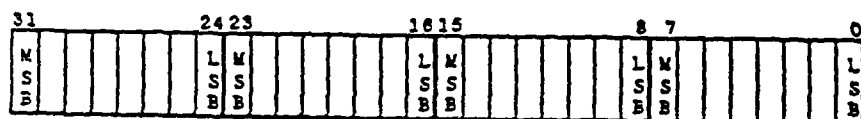bits within a byte

bits within a word

address A   address A+1

address of the word

bits within a longword

address A   address A+1   address A+2   address A+3

address of the longword

For bit field instructions, the bits are numbered differently. This numbering is used for record representation specifications

bit field bit numbers →

address A   address A+1   address A+2   address ...

address of the bit field
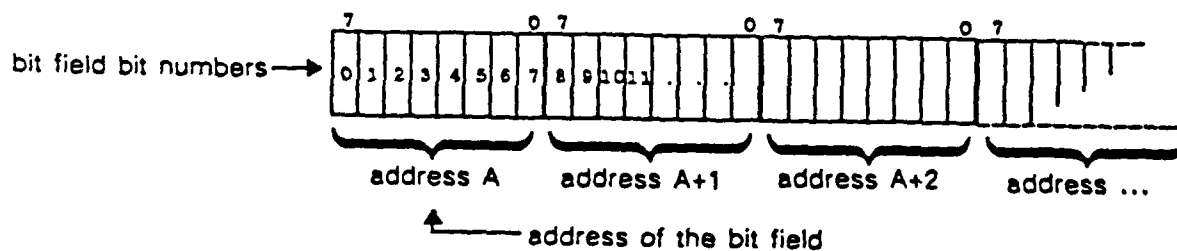
* Motorola. "2.3 Data Organization in Memory." *MC68020 32-Bit Microprocessor User's Manual*, p. 2-2. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

The only restrictions on record representation specifications are the following: if a component does not start and end on a storage unit boundary, it must be possible to get it into a register with one move instruction. On a machine such as the 386, this means such a field must fit into 4 bytes; on a MC68000 machine, where longwords start on even bytes, it must fit into 4 bytes starting on a word boundary. For example, the following specification is legal on a 386 target, but not on a MC68000 target machine.

```
for REC use record at mod 2;
   FIELD at 1 range 2 .. 25;
   -- Extraction of FIELD must start at byte 0 on MC68000. Byte 1,
   -- bit 25 is 34 bits beyond the start of byte 0 (34 bits > 4
   -- bytes, so cannot extract).
   -- On a 386, extraction can start at byte 1. Byte 1, bit 25 is
   -- 26 bits beyond the start of byte 1 (26 bits <= 4 bytes, so
   -- can extract).
end record;
```

Also, a component that is itself a record must occupy a power of 2 bits. Components that are of a discrete type or packed array can occupy an arbitrary number of bits, subject to the above restrictions.

Note that in the example above a size specification could be given,

```
for REC'size use 39;
```

but due to alignment, such a record would always take 5 bytes (i.e., 40 bits).

Address Clauses — Address clauses are supported for the following entities.
> objects
> entries

For information on converting INTEGER values to ADDRESS values, see also [Implementation Reference], Implementation Defined Attributes: 'REF, SYSTEM.ADDRESS'REF(N).

Change of Representation — Change of representation is supported.

package SYSTEM — For the specification of package SYSTEM, see also [Implementation Reference], Predefined Packages and Generics. This specification is also available on line in the file system.a in the release standard library. The pragmas SYSTEM_NAME, STORAGE_UNIT, and MEMORY_SIZE are recognized by the implementation, but have no effect. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling the SYSTEM package with altered values. Note that such a compilation will cause other units in the STANDARD

library to become out of date. Consequently, such recompilations should be made in a library other than standard.

System-Dependent Named Numbers —For the specification of package SYSTEM, see also [Implementation Reference], Predefined Packages and Generics.. This specification is also available on line in the file system.a in the release standard library.

Representation Attributes — The 'ADDRESS attribute is supported for the following entities.

- variables
- constants
- procedures
- functions

All other representation attributes are supported.

Representation Attributes of Real Types — These attributes are supported. See *[Implementation Reference]*, *Predefined Packages and Generics* for their values.

Machine Code Insertions — Machine code insertions are supported. See *[Implementation Reference]*, *Machine Code Insertions* and *Using VADS*, *Machine Code Insertions* for details.

Interface to Other Languages — See *[Implementation Reference]*, *Pragmas and Their Effects* and *Interface Programming* for details.

Unchecked Programming — Both UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION are provided.

Unchecked Storage Deallocations — Any object that was allocated may be deallocated. No checks are currently performed on released objects.

Unchecked Type Conversions — The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

18.12 Conversion and Deallocation

The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

There are no restrictions on the types with which generic function UNCHECKED_DEALLOCATION can be instantiated. No checks are performed on released objects.

18.15 Predefined Packages and Generics

The following predefined Ada packages given by RM Appendix C(22) are provided in the STANDARD .

- package STANDARD
- package CALENDAR
- package SYSTEM
- generic procedure UNCHECKED_DEALLOCATION
- generic function UNCHECKED_CONVERSION
- generic package SEQUENTIAL_IO
- generic package DIRECT_IO
- package TEXT_IO
- package IO_EXCEPTIONS
- package LOW_LEVEL_IO

- package MACHINE_CODE

```
package STANDARD is

    ...

    type INTEGER is range -2147483648 .. 2147483647;
    type SHORT_INTEGER is range -32_768 .. 32_767;
    type TINY_INTEGER is range -128 .. 127;
    type FLOAT is digits 15 range -1.79769313486231E+308 ..
                                    1.79769313486231E+308;
    type SHORT_FLOAT is digits 6 range -3.40282E+38 ..
                                        3.40282E+38;

    type DURATION is delta 1.00000000000000E-03
                     range -2147483.648 .. 2147483.647;

    ...

end STANDARD;
```

18.15.1 Specification of package SYSTEM

```
package SYSTEM
is
    type NAME is ( cadmus_munix );

    SYSTEM_NAME     : constant NAME := cadmus_munix;

    STORAGE_UNIT : constant := 8;
    MEMORY_SIZE     : constant := 16_777_216;

    -- System-Dependent Named Numbers

    MIN_INT         : constant := -2_147_483_648;
    MAX_INT         : constant := 2_147_483_647;
    MAX_DIGITS      : constant := 15;
    MAX_MANTISSA : constant := 31;
    FINE_DELTA      : constant := 2.0**(-31);
    TICK            : constant := 0.01;

    -- Other System-dependent Declarations

    subtype PRIORITY is INTEGER range 0 .. 99;

    MAX_REC_SIZE : integer := 64*1024;

    type ADDRESS is private;

    NO_ADDR : constant ADDRESS;

    function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
    function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
    function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
    function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
    function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
    function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
    function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
    function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

    function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
    function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
    function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
    function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
    function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
    function "+"(A: ADDRESS;
                 INCR: INTEGER) return ADDRESS renames INCR_ADDR;
    function "-"(A: ADDRESS;
                 DECR: INTEGER) return ADDRESS renames DECR_ADDR;

    pragma inline(ADDR_GT);
    pragma inline(ADDR_LT);
    pragma inline(ADDR_GE);
    pragma inline(ADDR_LE);
    pragma inline(ADDR_DIFF);
    pragma inline(INCR_ADDR);
    pragma inline(DECR_ADDR);
    pragma inline(PHYSICAL_ADDRESS);

private

    type ADDRESS is new integer;

    NO_ADDR : constant ADDRESS := 0;

end SYSTEM;
```

18.15.2 package CALENDAR

CALENDAR's clock function (in package CALENDAR.LOCAL_TIME located in the file calendar_s.a) uses the UNIX service routines GETTIMEOFDAY and LOCALTIME for getting the current time.

18.15.3 Package SEQUENTIAL_IO

Sequential I/O is currently implemented for variant records, but with the restriction that the maximum size possible for the record will always be written. This is also true of direct I/O. For unconstrained records and arrays, the constant, SYSTEM. MAX_REC_SIZE, can be set prior to the elaboration of the generic instantiation of SEQUENTIAL_IO or DIRECT_IO. For example, if unconstrained strings are written, SYSTEM.MAX_REC_SIZE effectively restricts the maximum size of string that can be written. If the user knows the maximum size of such strings, the SYSTEM.MAX_REC_SIZE may be set prior to instantiating SEQUENTIAL_IO for the string type. This variable can be reset after the instantiation with no effect.

18.16 Types, Ranges, and Attributes

Numeric Literals — VADS Ada uses unlimited precision arithmetic for computations with numeric literals.

Enumeration Types — VADS Ada allows an unlimited number of literals within an enumeration type.

Attributes of Discrete Types — VADS Ada defines the image of a character that is not a graphic character as the corresponding 2- or 3-character identifier from package ASCII of RM Annex C-4. The identifier is in uppercase without enclosing apostrophes. For example, the image for a carriage return is the 2-character sequence CR (ASCII.CR).

The type STRING — Except for memory size, VADS Ada places no specific limit on the length of the predefined type STRING. Any type derived from the type STRING is similarly unlimited.

Name of Attribute	Attribute Value of INTEGER	Attribute Value of SHORT_INTEGER	Attribute Value of TINY_INTEGER
FIRST	-2_147_483_648	-32_768	-128
LAST	2_147_483_647	32_767	127

Operation of Floating Point Types — VADS Ada floating point types have the attributes listed below.

Name of Attribute	Attribute Value of FLOAT	Attribute Value of SHORT_FLOAT
SIZE	64	32
FIRST	-1.79769313486231E+308	-3.40282E+38
LAST	1.79769313486231E+308	3.40282E+38
DIGITS	15	6
MANTISSA	51	21
EPSILON	8.88178419700125E-16	9.53674316406250E-07
EMAX	204	84
SMALL	1.94469227433160E-62	2.58493941422821E-26
LARGE	2.57110087081438E+61	1.93428038904620E+25
SAFE_EMAX	1022	126
SAFE_SMALL	1.11253692925360E-308	5.87747175411143E-39
SAFE_LARGE	4.49423283715578E+307	8.5075511654154E+37
MACHINE_RADIX	2	2
MACHINE_MANTISSA	53	24
MACHINE_EMAX	1024	128
MACHINE_EMIN	-1022	-126
MACHINE_ROUNDS	TRUE	TRUE
MACHINE_OVERFLOWS	TRUE	TRUE

Fixed Point Types — VADS Ada provides fixed point types mapped to the supported integer sizes.

Operations of Fixed Point Types — VADS Ada fixed point type DURATION has the attributes shown below.

Name of Attribute	Attribute Value for DURATION
SIZE	32
FIRST	-2147483.648
LAST	2147483.647
DELTA	1.00000000000000E-03
MANTISSA	32
SMALL	9.76562500000000E-04
LARGE	4.19430399902343E_06

```
FORE                          8
AFT                           3

SAFE_SMALL                    9.76562500000000E-04
SAFE_LARGE                    4.19430399902343E+06

MACHINE_ROUNDS                TRUE
MACHINE_OVERFLOWS             TRUE
```

18.17 Input/Output

The Ada I/O system is implemented using UNIX I/O. Both formatted and binary I/O are available. There are no restrictions on the types with which DIRECT_IO and SEQUENTIAL_IO can be instantiated, except that the element size must be less than a maximum given by the variable SYSTEM.MAX_REC_SIZE. This variable can be set to any value prior to the generic instantiation; thus, the user can use any element size. DIRECT_IO can be instantiated with unconstrained types, but each element will be padded out to the maximum possible for that type or to SYSTEM.MAX_REC_SIZE, whichever is smaller. No checking — other than normal static Ada type checking — is done to ensure that values from files are read into correctly sized and typed objects.

VADS file and terminal input–output are identical in most respects and differ only in the frequency of buffer flushing. Output is buffered (buffer size is 1024 bytes). The buffer is always flushed after each write request if the destination is a terminal. The procedure FILE_SUPPORT.ALWAYS_FLUSH (file_ptr) will cause flushing of the buffer associated with file_ptr after all subsequent output requests. Refer to the source code for file_io_body.a in the standard library.

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed before instantiating DIRECT_IO to provide an upper limit on the record size. The maximum size supported is 1024 * 1024 * STORAGE_UNIT bits. DIRECT_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as STRING where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before instantiating INTEGER_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such
as the maximum length of an input line and invalid file names. A test that
makes use of such values is identified by the extension .TST in its file
name.   Actual values to be substituted are represented by names that begin
with a dollar sign. A value must be substituted for each of these names
before the test is run.   The values used for this validation are given
below. Ada aggregate notation is used to denote long strings.   The
multiplication operator is meant to be overloaded to achieve repetition so
that e.g. 123*'A' is equivalent to (1..123 => 'A').

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	498 * 'A' & '1'
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	498 * 'A' & '2'
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	249 * 'A' & '3' & 249 * 'A'
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	249 * 'A' & '4' & 249 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	496 * '0' & "298"

TEST PARAMETERS

Name and Meaning	Value
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	494 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 249 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 249 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	479 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	X}]\%!@#$&~Y
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	WILD-CHAR*.NAM
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	000000.0

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	2147484.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD-CHARACTER*^/%
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	AAA/AAA
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	000000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-2147484.0
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	499
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

TEST PARAMETERS

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 494*'0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 492*'0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 497 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	Tiny_Integer
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- B28003A: A basic declaration (line 36) wrongly follows a later declaration.

- E28005C: This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ARG.

- C34004A: The expression in line 168 wrongly yields a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

- C35502P: Equality operators in lines 62 & 69 should be inequality operators.

- A35902C: Line 17's assignment of the nomimal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may in fact raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

WITHDRAWN TESTS

- C35A03E, C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

- C37213H: The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, C37215H: Various discriminant constraints are wrongly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: 'STORAGE_SIZE is wrongly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

- A74106C, C85018B, C87B04B, CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur in lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere).

- BC3105A: Lines 159..168 are wrongly expected to be illegal; they are legal.

- AD1A01A: The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 & 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.